Predicting time series - NODE vs Reservoir computer

Albin Steen, Viktor Olsson, Simone Piccioni, Ole Fjeldså, and Zachary Tio
(Chalmers Tekniska Högskola)
(Dated: May 30, 2024)

Today, diverse data varying in time and space is crucial across fields like medicine, finance, and electronics. Despite the significance of these fields, handling data in general as well as chaotic data poses challenges for prediction. Controlling and accurately predicting data is paramount for advancing work in these critical domains. One of the most prominent modern methods to predict data and especially chaotic data is Reservoir computing. This research explores how a rather new method in Neural ODE compares to a reservoir computer in its capability of predicting both more stable and chaotic data. This will be executed by implementing a reservoir computer as well as a neural ODE and then training as well as testing them on data from a Van der Pol system, Lotka-Volterra system and a chaotic system in a Lorenz attractor. This implementation aims to give insight into the performance of both Neural ODE and a Reservoir computer as well as how the two compare.

I. INTRODUCTION

Prediction of data is something that has become vital in many areas of modern society. Furthermore, chaotic data, characterized by variability over time and space and unpredictability, holds a lot of relevance across several critical fields, including medicine, finance, and electronics. In medicine, the patient's data is influenced by various biological factors and environmental variables. This means challenges for correct diagnosis, treatment planning (e.g. dose planning [1]), and disease prediction. Similarly, in finance, there are market fluctuations, influenced by factors such as economic indicators, geopolitical events, etc. Furthermore, these create complex and difficult datasets that require techniques for risk management and investment decision-making [2][3]. In the field of electronics, the design and optimization of electronic systems are very much reliant on understanding and managing interactions between components, circuits, and signals [4]. All of these contribute to the unpredictable nature of electronic data. In each of these fields, the ability to handle and predict chaotic data is of great importance for making informed decisions, driving innovation, and ultimately advancing the respective industries.

A. Background

1. *Neural ODE*

Neural Ordinary Equations (NODEs) are a new type of Neural Networks (NN) that combine standard NN architecture and dynamical systems. Despite the interesting mathematical properties associated with it, they have proven to have several technical advantages compared to traditional architectures, such as constant memory usage which does not require saving all the intermediate steps as in classical backpropagation, improved accuracy with a fewer number of parameters as compared to traditional residual networks, naturally model continuous time variables, and more.

A natural way to introduce NODEs is in fact starting from the general model of a residual network [5]. Let's look at what's the equation for one layer in a residual network:

$$h_{t+1} = h_t + f(h_t, \theta_t). \tag{1}$$

Here, t denotes the discrete time step, h_t represents the hidden state at time t, f is a neural network function parameterized by θ_t, and θ_t represents the parameters of the neural network at time t.

Let's imagine that the function f is the same for all the layers and that we add many layers and take very small steps. By doing this we will reach a limit where hidden state dynamics is represented by an ODE:

$$\frac{dh(t)}{dt} = f(h(t), t, \theta), \tag{2}$$

$h(t)$ represents the hidden state at continuous time t, f is a neural network function with parameters θ, and the continuous time dynamics are described by this differential equation.

Starting from an initial input layer $h(0)$, the output layer $h(T)$ is defined as the solution to the ODE initial value problem at some final time T. The solution to the ODE can be computed using a black-box differential equation solver, which evaluates the hidden unit dynamics f as necessary to determine the solution with the desired accuracy.

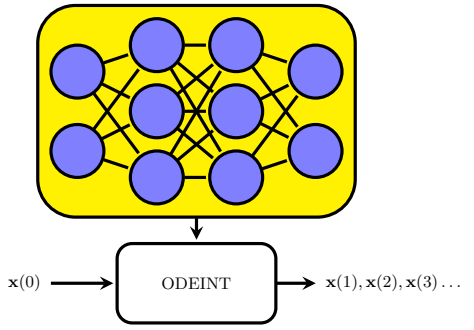

FIG. 1: The typical structure for a neural ODE,

The main technical difficulty in training continuous-depth networks is performing reverse-mode differentiation (backpropagation) through the ODE solver.
In fact, calculating the derivatives of the loss function with respect to the weights of the neural network, is not as straight-forward, considering that the output has the following form:

$$\hat{y} = \text{ODEINT}(x_0, f_{\text{NN}}(x, t, w)) \tag{3}$$

where $f_{\text{NN}}(x, t, w)$ is the neural network with weights w, x_0 is the input, and ODEINT is some integrator of our choice.

Gradients are computed using the adjoint sensitivity method, which solves an augmented ODE backward in time, by means of an integral:

$$\frac{\partial L}{\partial w} = -\int_{t_1}^{t_0} a(t)^T \frac{\partial f_{\text{NN}}(x, t, w)}{\partial w} dt \tag{4}$$

where the quantity $a(t) := \partial L/\partial x$ is called adjoint-state, and describes the rate of change of the loss L, with respect to one of the continuous layers $x(t)$. More details on the method can be found in [6].
It's worth mentioning that the method scales linearly with problem size has low memory cost, and controls numerical error.[6]

2. *Reservoir Computing*

Moving on to methods for time series prediction, Reservoir Computing was introduced not long ago as a new approach to training Recurrent Neural Networks. The main vital theory behind reservoir computing is to use techniques that are designed to model complex dynamic systems by employing so-called fixed-weight recurrent networks.

Furthermore, to explain it in more detail, these types of networks use datasets to produce activation states, which are then in turn used to train output weights. These weights give descriptions of the original data model's dynamics. The most vital part of Reservoir Computing is the Reservoir, functioning as a "black-box" model with fixed weights, and the Readout, usually a simple linear classifier layer. Crucially, these methods offer inherent memory effects due to the recurrent connections within the Reservoir.[7]

Furthermore, to train the linear output layer usually some type of linear regression is employed. For this project ridge regression is used. Ridge regression is used to train the output weights by minimizing the error between the predicted and real outputs while adding a regularization term to prevent overfitting [8]. The reservoir states created during training are used as inputs to the ridge regression. By solving the regularized least squares problem, it will adjust the output weights to best fit the training data.

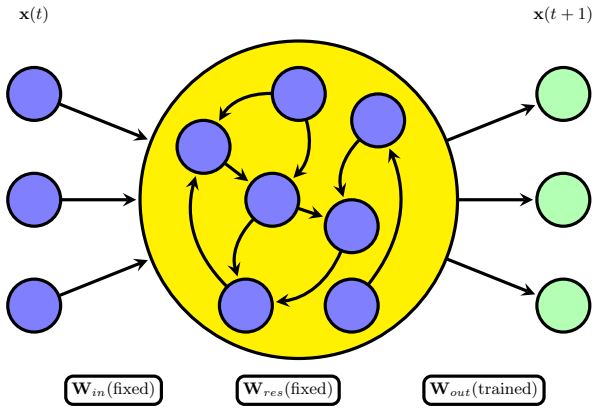

FIG. 2: The typical structure for a reservoir computer, every node in the input is connected to every node in the reservoir, and every node in the reservoir is connected to every node in the output. The reservoir is sparsely self connected between time steps.

3. *Van der Pol system*

The first more normal and less chaotic data set is derived from a Van der Pol oscillator. The dynamics of the Van der Pol oscillator are described by a second-order nonlinear differential equation:

$$\frac{d^2x}{dt^2} - \mu(1 - x^2)\frac{dx}{dt} + x = 0, \tag{5}$$

where x represents the displacement of the oscillator, t denotes time, and μ is a parameter governing the nonlinearity and damping strength of the system. This equation describes the interplay between the restoring force, damping, and nonlinear damping term $-\mu(1 - x^2)\frac{dx}{dt}$, which causes the oscillations to exhibit limit cycle behavior.

The Van der Pol oscillator exhibits dynamic behavior. At small values of μ, the system behaves like a harmonic oscillator, showing sinusoidal oscillations. However, as μ increases, the oscillations become more complex, eventually leading to a stable limit cycle. This limit cycle

characterizes the self-sustained oscillations that defines the Van der Pol oscillator. [9]

4. *Lotka-Volterra system*

The second system for this research is the Lotka Volterra system. The Lotka-Volterra system consists of a pair of coupled first-order nonlinear differential equations, capturing the dynamics of predator and prey populations over time:

$$\begin{aligned} \frac{dx}{dt} &= \alpha x - \beta xy, \\ \frac{dy}{dt} &= \delta xy - \gamma y, \end{aligned} \tag{6}$$

where x represents the population of prey species, y denotes the population of predator species, and α, β, γ, and δ are parameters governing birth rates, predation rates, natural mortality, and conversion efficiency, respectively. These equations describe the processes of reproduction, predation, and mortality that drive the dynamics of ecological communities. [10]

5. *Lorenz system*

Regarding chaotic data, the study predominantly focuses on utilizing chaotic data derived from the Lorenz system. The Lorenz system is described by a set of three first-order nonlinear differential equations, which describe the evolution of three-dimensional dynamical systems:

$$\begin{aligned} \frac{dx}{dt} &= \sigma(y - x), \\ \frac{dy}{dt} &= x(\rho - z) - y, \\ \frac{dz}{dt} &= xy - \beta z, \end{aligned} \tag{7}$$

where x, y, and z represent the state variables, t denotes time, and σ, ρ, and β are parameters governing the system's behavior. These equations, eq. 7, show the interplay between convection, rotation, and dissipation in a simplified atmospheric model, giving rise to the chaotic trajectories that characterize the Lorenz attractor.

The Lorenz system has many types of chaotic factors, including sensitive dependence on initial conditions, aperiodic dynamics, and the presence of strange attractors. Small perturbations in the initial conditions can lead to different trajectories over time, showing the sensitivity of chaotic systems to very small changes. [11]

B. Purpose

The primary purpose and objective of this project and research is to conduct a comparative analysis of the predictive capabilities for both chaotic and more predictable data between an NODE and a reservoir computer. This investigation aims to ascertain whether the NODE exhibits superior performance compared to reservoir computers. The evaluation will primarily focus on predictive accuracy and performance, while also considering the efficiency of model training.

Ultimately, the findings hold the potential to pave the way for the implementation and utilization of these models in handling and accurately predicting data within critical domains such as medicine, economics, and electronics. Such advancements could significantly help decision-making processes in critical situations within these important environments.

II. METHODS

There are two sections of methods for this project. The first is to generate data for the three systems that the project explores. The second section is about the "predictors", ie implementing a reservoir computer and a NODE, training them, and letting them do predictions.

A. Generating Data

First, the data sets are generated by iterating the dynamics for each of the three systems. The Van der Pol system is governed by the second-order equation mentioned in the background section. But to generate data from it, the equation is transformed into two dimensions according to this system:

$$\begin{aligned} \dot{x} &= \mu \left(x - \frac{1}{3}x^3 - y \right), \\ \dot{y} &= \frac{1}{\mu} x. \end{aligned} \tag{8}$$

For the purpose of this project, the value of μ was set to 1, in order to obtain a a dynamic dominated by the presence of a limit cycle. Then a single trajectory with starting point $x = y = 1$ was generated, by numerically integrating eq. 8, with a time step of $\delta t = 0.1$.

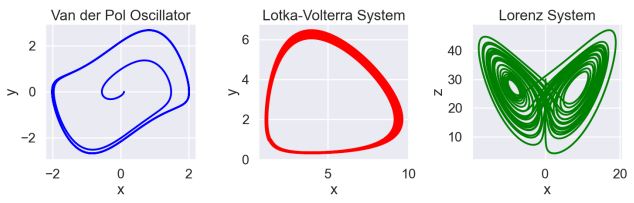

FIG. 3: A visual representation of the three investigated systems.

The next system is the Lotka Volterra system, this system is governed by the equations presented in the back-

ground section, eq. 6. Parameter values are $\alpha = 1/4, \beta = 1/2, \delta = 1/2, \gamma = 1/4$. As before a single trajectory was generated with starting point $x = y = 1$, by numerically integrating the differential equation with a time step of $\delta t = 0.1$.

Lastly, we have the chaotic system in a chaotic Lorenz attractor. With the parameter values $\sigma = 10, \rho = 28, \beta = 8/3$ we can get a chaotic attractor for the system. Again by numerical integration (with a smaller time step $\delta t = 0.02$) and by choosing $x = y = z = 1$ as the starting point, the single trajectory was generated.

Furthermore, the time series produced for the three systems is split into two segments, where the first segment is employed for training reservoir computers or NODEs. The predictive capabilities of these models are then evaluated on the second segment of the time series.

A visual representation of the three systems can be seen in figure 3.

B. Implementing Predictors

There are two predictors, the reservoir computer and the NODE.

1. Implementing a Reservoir computer

First, to implement the reservoir computer, it's not too complicated and it can be implemented from scratch without using any big packages like Pytorch.

The basis of implementation is that you need an initialization of input, reservoir, and output weights[12]. The input weights are initialized with a uniform distribution between -0.3 and 0.3. The reservoir weights are also uniformly initialized between -1 and 1 in a sparse matrix with only 8% of the reservoir weights being non-zero. The reservoir weight are then also scaled by the singular value multiplied with a parameter $\rho = 0.001$. The output weights are put to zero and then calculated after executing the training using a linear regression method which for this project specifically is ridge regression as previously mentioned.

After the model is built it can be initialized and then the first part of the generated data can be fed to the model. To train the following equation is iterated:

$$r_i(t+1) = g\left(\sum_j w_{ij} r_j(t) + \sum_{k=1}^{N} w_{ik}^{\text{in}} x_k(t)\right) \tag{9}$$

Here g is the activation function which is sigmoid, $r_i(t)$ is the reservoir state for node "i" at a time "t", w represents weights, and $x_k(t)$ is the input in dimension "k" at a time "t". By iterating this function for all the time steps in the input data a series of reservoir states r is generated for all the time steps. This series of states is then used with ridge regression to train the output weights.

After training, the first value of the second part ie testing part, of the generated data is fed to the model. Predictions can then be produced by first using equation 9 and then the equation below to produce an output/prediction:

$$O_i(t+1) = \sum_{j=1}^{M} w_{ij}^{\text{out}} r_j(t+1) \tag{10}$$

Moreover the first prediction of the $x_k(t)$ in equation 9 is exchanged with the output $O_i(t)$ and in the end the model will then loop over equation 9 and then equation 10 for the time that is defined by how long the testing part of the data is. In the end, a predicted time series is produced.

2. Implementing neural ODE

Moving on, the implementation of the NODE was done by using pytorch and more specifically the package **"pydiffeq"** [13].

A deep fully connected neural network, with 8 Linear layers (with the following number of neurons: $3, 2^8, 2^8, 2^9, 2^9, 2^8, 2^8, 3$) and ReLU activation functions, has been used to approximate the original function that define the various dynamical systems considered.

For the ODE integrator, the built-in function *odeint_adjoint*, of the package pydiffeq (which by default uses the classical Runge-Kutta method, RK4), has been used. This function naturally performs the adjoint sensitivity method and it's compatible with Pytorch's optimizers and loss functions, allowing to keep the same typical code structure of Pytorch.

Compared to the reservoir less data to actually train the network, was used. This due to the fact that using too long trajectories, usually results in extremely long times for a single epoch and worse test results, compared to shorter trajectory and same number of epochs. For this reason, a standard trajectory size of 1000 points has been set (unless specified), as it has been seen that it would generate slightly better results.

During a single epoch, all the points on the training trajectory (except the last one), are used as the starting point of an initial value problem, from which the next point is obtained and used to calculate the MSE loss, and then perform backpropagation to update the weights with the ADAM optimizer (with a learning rate of in the first 1000 iterations 10^{-2} and for the successive ones 10^{-3}).

The total number of epochs depended on the system

that was being considered:

- Lotka-Volterra: 300 epochs
- Van der Pol: 200 epochs
- Chaotic Lorenz: 2000 epochs

Once the training had been completed, the first value of the test trajectory was given as the starting point, while the best scoring network over all the epochs, was the function for the IVP that would generate the entire test trajectory.

III. RESULTS

The results will be presented by looking at the predictions for one of the components within each system as well as a trajectory plot for the chaotic lorenz attractor, the complete images are a bit wide and are therefore presented in Appendix A.

A. Van der pol

First of, looking at figure 4, for this time series both the reservoir and the NODE works really well. Both the methods manages to predict the time series in near perfection and we see no real visible divergence in any place of the whole of the time series that the methods were tested on.

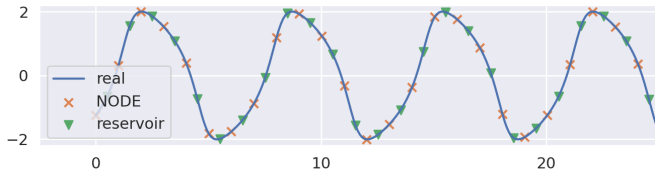

FIG. 4: Comparisons between NODE and reservoir computer over the Van der Pol dataset. The complete one can be found in Appendix figure 8.

B. Lotka-Volterra

In contrast to the Van der Pol results, when viewing figure 5, it becomes apparent that there are some differences between the performances of the two methods.

It seems that both the methods predict the general shape of how the component changes very well. Although there is an offset between the real-time series and the methods. For the reservoir computer, this offset seems to start already after about 20 seconds where the reservoir seems to lag a bit behind the original series, this lag then grows as time progresses. As figure 5 suggests, there is a "complete" separation/divergence at around 73.7 seconds.

Furthermore, for the NODE there is also a bit of an offset but the NODE seems to predict the values a bit earlier than when they actually happen ie it does not lag behind but makes a bit of an early prediction. But this offset is much smaller than for the reservoir computer and also it first appears much later than for the reservoir. The divergence point is measured to be at around 150s ie double the time for the reservoir.

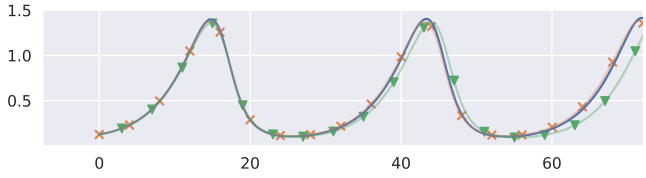

FIG. 5: Comparisons between NODE and reservoir computer over the Lotka-Volterra dataset. The complete one can be found in Appendix figure 9.

C. Chaotic Lorenz attractor

For this time series, we get the most apparent results yet. Figure 6 shows that neither the reservoir computer nor the NODE predicts the chaotic time series perfectly or even very well. But in contrast to the results for the Lotka-Volterra time series, for the chaotic data the reservoir computer clearly performs a lot better then the NODE.

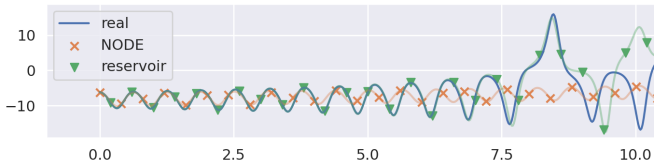

FIG. 6: Comparisons between NODE and reservoir computer over the Chaotic Lorenz dataset. The complete one can be found in Appendix figure 10.

There seems to be a divergence in the NODE prediction almost immediately which is strengthened by the fact that the calculated divergence was 2.24 seconds. The Reservoir does not manage a huge time before diverging with 7.5 seconds, but it is significantly better than the NODE. Furthermore, according to figure 10, the reservoir even after the divergence seems to have somewhat of the same behavior as the real-time series while the NODE is completely different.

This point is reinforced by the following figure:

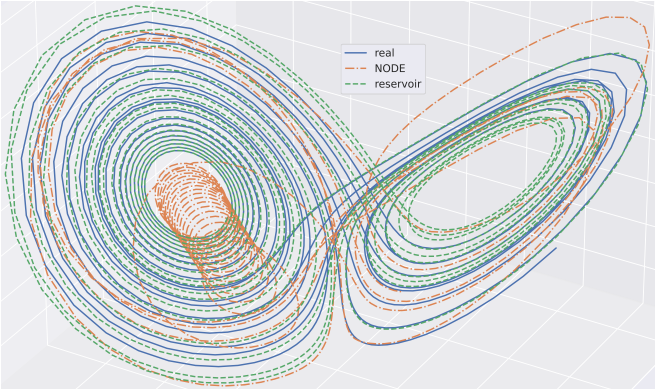

FIG. 7: Trajectories for the chaotic Lorenz attractor. Both NODE and Reservoir are getting to the Lorenz attractor but the NODE does considerably worse.

This figure shows the complete trajectories in three dimensions. In this figure, it's shown that the reservoir although it does not predict the time series very well, follows the behavior of the Lorenz attractor quite well the whole way. In contrast, the NODE behaves very differently and seemingly out of sync. This figure is also one of the better trajectories that the NODE managed during all the simulations for the Lorenz attractor time series.

IV. DISCUSSION

The task of predicting time series is inherently hard. Here we compared these two methods that try to accomplish this, with two substantially different philosophies.

On one hand, the reservoir computer, which by embedding the low-dimensional input into a high-dimensional space, tries to capture the local dynamic of the series, by approximating the Jacobian and using it to predict the next point.

On the other hand NODE, which tries to figure out the overall dynamic that generated the series, by approximating the function that defines the system, with a neural network.

A difference between these two methods is also present in the training, in fact while the reservoir computer requires longer trajectories to perform well, the NODE can be trained on much shorter trajectories and still get good results (provided that the system is not too chaotic). For example for the Lotka-Volterra system, the reservoir computer has been trained over a trajectory consisting of 4000 points, while the NODE only on 2000, and as can be seen in figure 9 the NODE outperforms the reservoir computer.

The training times associated with the methods, also differ significantly, in fact the reservoir computer, which uses ridge regression, only needs one single iteration over the whole ordered dataset.

The NODE instead, requires several iterations over the training-trajectory to be able to predict the correct dynamic. For example for the Lorenz system, NODE required more than 1500 iterations, to give acceptable results.

In terms of the prediction, the trained NODE only requires the first point of a trajectory to be able to predict up to any arbitrary time in the future, while the reservoir computer usually also requires that the memory gets initialized over previous parts of the series. Moreover, NODE showed more stability in the long-term prediction (provided again that the system isn't chaotic), as it can be seen for example, in which even if both predictions slowly shift away from the real trajectory, the reservoir drifts away at a much faster rate compared to NODE.

Finally coming to the Lorenz system, which was used to test the predictive abilities of these methods on chaotic time series, showed that NODE generally fails at capturing the dynamic, only being able to predict only a few seconds in the future (see also [14] and [15]). While the reservoir computer, if properly set up, can predict up to 25s on the same time series.

This could potentially arise from the fact that when the system is chaotic, trying to capture the whole dynamical system, from a single trajectory is almost an impossible task, and hence the method only manages to capture shorter pieces of the trajectory, by means of functions that differ from the original one (equation 7).

V. CONCLUSION

Both the reservoir computer and the NODE can predict less chaotic and more regular time series very well. For chaotic data, the NODE struggles a lot more compared to the Reservoir computer.

The reservoir computer performs better with the chaotic data because it captures only the local dynamic of the system, while the NODE performs worse as it tries to get the global dynamic. Thus, reservoir computing is preferred for chaotic data.

Unless the parameters for the reservoir computer are fine-tuned, the NODE has a slightly more robust behavior for the less chaotic data. Therefore, it could be argued that for this data NODE is preferred.

While the cost of training a system should be taken into consideration, when actually running a pretrained model the runtime of either model is negligible. This would make NODE a good choice for some use cases. For example as part of a control system where it only

needs to predict a short time into the future.

Future work: A more methodic way to train NODEs could be explored, especially over chaotic datasets. Moreover, other techniques, related to NODEs, can be analyzed, such as *generative latent function time-series model*, see [6], or even more modern techniques such as *semi-implicit NODEs*, see [15].

VI. CONTRIBUTIONS

Every member of the project contributed to everything in the project and every member takes responsibility and stands behind the content of this report and the project itself.

VII. ACKNOWLEDGEMENTS

We would like to acknowledge Yu-Wei Chang, our supervisor for his support and advice.

[1] J. Lu, K. Deng, X. Zhang, G. Liu, and Y. Guan, Neural-ode for pharmacokinetics modeling and its advantage to alternative machine learning models in predicting new dosing regimens, iScience **24**, 102804 (2021).
[2] J. Li, W. Chen, Y. Liu, J. Yang, D. Zeng, and Z. Zhou, Neural ordinary differential equation networks for fintech applications using internet of things, IEEE Internet of Things Journal , 1 (2024).
[3] A. Nguyen, S. Ha, and H. Thai, Phase space reconstructed neural ordinary differential equations model for stock price forecasting, SSRN http://dx.doi.org/10.2139/ssrn.4817927 (2024).
[4] S. Pepe, J. Liu, E. Quattrocchi, and F. Ciucci, Neural ordinary differential equations and recurrent neural networks for predicting the state of health of batteries, Journal of Energy Storage **50**, 104209 (2022).
[5] K. He, X. Zhang, S. Ren, and J. Sun, Deep residual learning for image recognition, (2015), arXiv:1512.03385.
[6] R. T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud, Neural ordinary differential equations (2019), arXiv:1806.07366 [cs.LG].
[7] L. Melandri, *Introduction to Reservoir Computing Methods*, Ph.D. thesis, University of Bologna (2014).
[8] A. E. Hoerl and R. W. Kennard, Ridge regression: Biased estimation for nonorthogonal problems, Technometrics **12**, 55 (1970), https://www.tandfonline.com/doi/pdf/10.1080/00401706.1970.10488634.
[9] Wikipedia contributors, Van der pol oscillator (2024), [Online; accessed 2-May-2024].
[10] Wikipedia contributors, Lotka–volterra equations (2024), [Online; accessed 2-May-2024].
[11] Wikipedia contributors, Lorenz system (Accessed 2024), accessed on 2024-04-22.
[12] M. Lukoševičius, A practical guide to applying echo state networks, in *Neural Networks: Tricks of the Trade: Second Edition*, edited by G. Montavon, G. B. Orr, and K.-R. Müller (Springer Berlin Heidelberg, Berlin, Heidelberg, 2012) pp. 659–686.
[13] R. T. Q. Chen, torchdiffeq (2018).
[14] A. Axelsson, Neural odes (2022).
[15] H. Zhang, Y. Liu, and R. Maulik, Semi-implicit neural ordinary differential equations for learning chaotic systems, 37th Conference on Neural Information Processing Systems (NeurIPS 2023) (2023).

Appendix A:
Prediction results of component comparisons

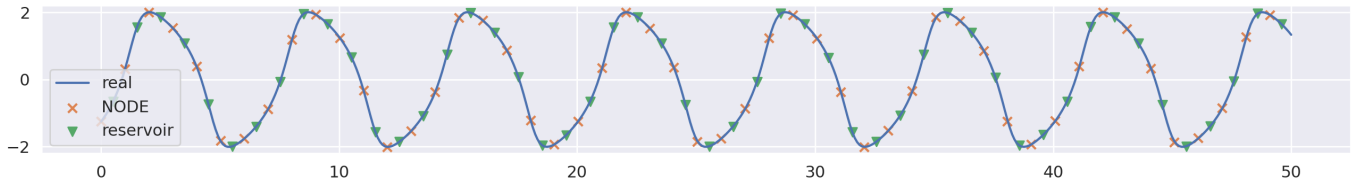

FIG. 8: NODE and Reservoir on the first component of the Van der Pol system.

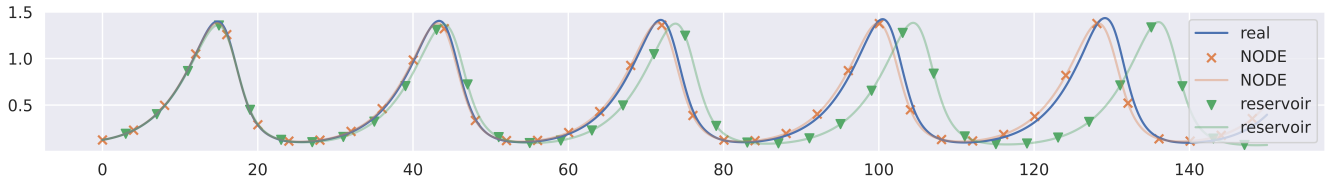

FIG. 9: NODE and Reservoir on the first component of the Lotka-Volterra system. NODE predicted time:150s, Reservoir predicted time:73.7s.

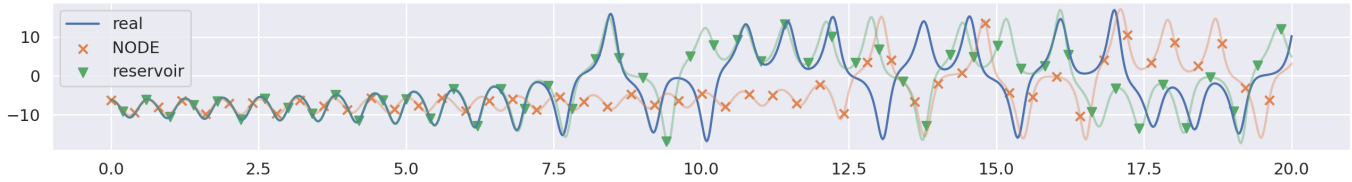

FIG. 10: NODE and Reservoir on the first component of the Lorenz system.
NODE predicted time:2.24s, Reservoir predicted time:7.5s.

Appendix B:
GitHub repository link

TIF360 group 15